SAWE Paper No. 3355

Reference Models for
Structural Technology Assessment and Weight Estimation

Jeff Cerro
Zoran Martinovic
Exploration Concepts Branch
NASA Langley Research Center

Lloyd Eldred
Swales Aerospace Corporation
Hampton VA

For Presentation at the
64th International Conference
of the
Society of Allied Weight Engineers, Inc.
Annapolis Maryland   16-18th May, 2005

I Overview

Previously the Exploration Concepts Branch of NASA Langley Research Center has developed techniques for automating the preliminary design level of launch vehicle airframe structural analysis for purposes of enhancing historical regression based mass estimating relationships ref [1]. This past work was useful and greatly reduced design time, however its application area was very narrow in terms of being able to handle a large variety in structural and vehicle general arrangement alternatives. Features of that work such as the utilization of Object Oriented JAVA Programming and the incorporation of flexible commercial FEA and commercial structural design software are retained in this continuing work, but a new emphasis has been placed on making the integrating JAVA modules much more generic. The goal has been to develop a library of JAVA modules which when placed in the desired sequence facilitate the automated structural sizing of a greater variety of component and vehicle systems. The finite element procedures wrapped by JAVA routines now trend towards being more generic in the sense that the routine inputs are not as much design and FEA program specific as they are design and FEA process specific. A later goal in this analysis system development would be to arrive at a working group defined set of JAVA Interface Classes that describe input and output required for particular stages of analysis of automated structural design. Along with standardized input/output parameters there would also be a set of standard data processing functions which are then useful to the structural designer in providing the flexibility required for designing numerous parts, sub-assemblies, and full vehicle configurations. In JAVA programming terminology these Class definitions become generic Interfaces which are then implementable at any corporate or academic organization utilizing internal and possibly proprietary procedures. Model data may be exchanged between these organizations and will be processable by any of the organizations which have implemented the defined standard Interface. Similar work is ongoing in the area of Simulation Based Acquisition (SBA) via the Simulation Interoperability Standards Organization (SISO) and particularly in the area of integrating distributed simulations by the High Level Architecture – Commercial off the shelf Simulation Package Interoperation Forum (HLA-CSPIF) ref [2]. For those more interested in preliminary design in a collaborative environment the NAVY NAVSEA division has pursued similar themes of modularity and multi-disciplinary interoperability by utilizing a CORBA and IDL (Interface Definition Language) based approach to Simulation Based Design (SBD) ref [3]. A quote from reference [3] shows the great utility of implementing a formal Simulation Based Design approach.

> “ High Potential was evidenced by the ability to integrate high-fidelity modeling and simulation tools to provide insight into overarching system-level performance attributes and the ability of the integration process itself to promote informal collaboration between various domain experts.”

These too are the features being encouraged and developed within the Exploration Concepts Branch to enable functionally and organizationally collaborative multidisciplinary design for the purposes of defining and building vehicle elements which best help us achieve the nations vision for manned space exploration ref [4].

Implementation of the analysis approach presented herein also incorporates some newly developed computer programs. Loft [5], is a program developed to create analysis meshes and simultaneously define structural element design regions. A simple component defining ASCII file is read by Loft to begin the design process. HSLoad [6] is a Visual Basic implementation of the HyperSizer Application Programming Interface, ref [7], which automates the structural element design process. Details of these two programs and their use are explained in this paper.

A feature which falls naturally out of the above analysis paradigm is the concept of “reference models”. The flexibility of the FEA based JAVA processing procedures and associated process control classes coupled with the general utility of Loft and HSLoad make it possible to create generic program template files for analysis of components ranging from something as simple as a stiffened flat panel, to curved panels, fuselage and cryogenic tank components, flight control surfaces, wings, through full air and space vehicle general arrangements.

II LOFT: Automated Mesh Generation

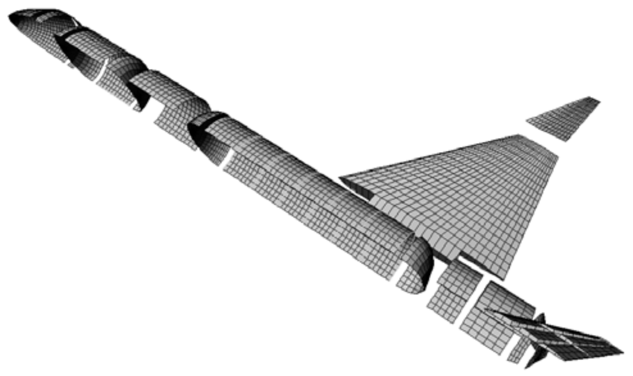


Figure 1. A Single Stage to Orbit vehicle mesh generated by *Loft*. *This exploded half view was used to illustrate the 18 different components in the model.*

Parametric mesh generation for stiffened shell aerospace vehicles was examined for application in an automated design environment. Schemes using both I-DEAS [8] parametric object geometries and simple “program file” script language driven generators were tried with mixed success. The critical shortcomings of these approaches were in mesh generation; I-DEAS was inconsistent about element generation ordering and no working method of automatically labeling elements based on their location in the vehicle was developed. A conceptual-level finite element model of most aerospace vehicles is not complex. For current analysis purposes generation of a three dimensional FE mesh representing a fuselage, tanks, bulkheads, and wings did not require a powerful CAD/CAM program. Thus it was decided to develop a very focused mesh generation code that would meet these specialized needs much better. Following the successful use of a prototype code that generated the mesh for a rib/spar stiffened trapezoidal wing, a substantially expanded code was developed to generate a conceptual-level finite element mesh for arbitrary aerospace vehicles.

Loft is the name of this in house developed mesh generator. Loft reads a text file containing a list of vehicle components and their dimensions. It generates an annotated mesh in

I-DEAS universal, NASTRAN bulk, and/or VRML 1.0 file formats. The input file is easy to read and compact; Figure 1 shows a full single stage to orbit vehicle model with 18 components including two frame stiffened fuel tanks, wing, winglet and vertical tail. This mesh was specified in less than one hundred lines of text with each line consisting of a single parameter name and corresponding value (typically 20 characters or less). This simplicity of input to complexity of mesh mapping is possible because the program is very specialized to these aircraft-like stiffened skin structures and because there is no need to specify many dimensions and positions unless the program default needs to be overridden.

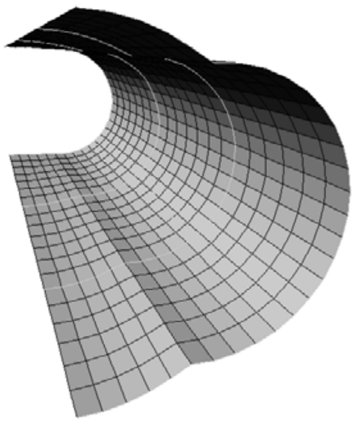


Figure 2. A Stiffened Fuselage Section extruded between two different User Defined Compound Curves in *Loft*.

The simplest geometric entity in Loft is called a "curve". Loft has a built in library of simple curves such as circles, squares, breadboxes, and filleted squares. Loft also supports two schemes for user definition of these primitive shapes: a linear interpolation between specified coordinates, and a complex combination curve made up by combining any previously defined curves. All of these shapes can be stretched and deformed to produce just about any imaginable cross section.

Loft then can create a three dimensional object from a curve in two ways. First, it can linearly extrude that curve to another curve. This will create a fuselage section, tank barrel, or conical thrust structure. Figure 2 illustrates a section extruded between two different user defined compound curves. Note the white lines that indicate the location of beam elements representing stiffening frames. Loft can alternatively extrude a curve to a single point to create a nose, dome, or bulkhead.

A wing object is somewhat more complicated. A wing is extruded based on a desired four digit NACA airfoil, with internal ribs and spars and optional carry-through. Support is also provided for partial wing objects representing control surfaces (or wings with a control surface cut out) and body flaps. Only trapezoidal wing planforms are supported, although complex shapes can be built up from trapezoidal sections.

A Loft input deck is a simple text file (see Table 1 for a fragment of the file that created the mesh in Fig. 1). Generally, the designer specifies the nose of the vehicle first giving its name, cross-sectional shape, length, diameter in the vertical and horizontal directions, mesh and sizing density values. (At the sizing stage of the analysis a panel is generally made up of multiple finite elements). Then the designer will sequentially specify the components moving aft on the vehicle. Loft will automatically use the settings for the previous component to size and position the next one. The designer only has to specify any values that are different than that default.

```
# Our nose
object dome Nose
curve1 sc
c1_xscale 15.589
c1_yscale 15.589
length -36
taper para
nodes_circ 21
nodes_axial 20
droop line
zdroop 8
```

Table 1: *Loft* file data to create Orbiter Nose

Some axial settings such as length will need to be specified for each component, but circumferential settings will generally not be changed unless the diameter expands or tapers.

This scheme of default generation has the extremely useful side effect of making the mesh very robust to changes. If, for instance, the volume of one of the tanks needs to be increased part way through the analysis, the single line that specifies its barrel length can be changed. Loft will automatically move all of the later components aft by this change and all objects will still automatically stitch together correctly and have the same labels.

Loft generates a substantial amount of annotation for each component as it is meshed. The user has control over much of this annotation. Every element in a component is marked by the names of its associated physical and material properties. The physical property name starts with the user specified object name, such as “FWD LH2 TANK”. In a few cases, such as a wing object, additional text is added to the physical property name to further differentiate the elements. For wings those text additions are “RIB”, “SPAR”, “SKIN UPPER” and “SKIN LOWER”. The names of the element’s material properties are used to indicate positions on the object such as “AXIAL 3 CIRC 4” for extruded domes and barrels or “SB 0 CB 3” for wing panels. In this shorthand notation, SB corresponds to “Spanwise Bay”, and CB to “Chordwise Bay”. These panel markings are used to control the size of the HyperSizer “components” discussed later and are explicitly controlled by the user specifying the component counts they desire in each direction. It should be noted at this point that the actual values for the physical and material property settings are not set by Loft. It will save dummy values for these settings. The actual property values are defined at the HyperSizer stage discussed below.

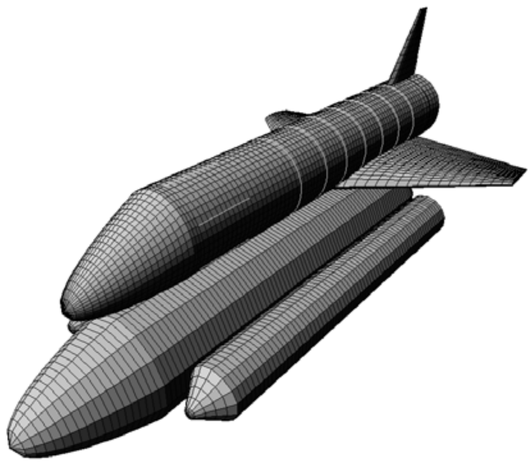


Figure 3. A simple representation of a shuttle-like launch stack built from a half model orbiter, half model external tank, and single solid-rocket booster.

In addition FEA entity groups are created that collect all of the nodes in the component, all of the elements, the nodes along the axis of symmetry, etc. For wings, additional groups collect nodes on the carry-through, and nodes at the trailing edge and leading edge root and tip spars. Further, the designer can add the nodes or elements of any component to arbitrarily specified collections. All of these collections are available as I-DEAS groups. They make it easy to apply loads and boundary conditions to the mesh in an automated fashion.

Loft has a wide variety of advanced features. These include a large collection of three dimensional translational, rotational, scaling, and warping features. A large scale cut and paste support allows for both cloning of components as well as multiple configuration support in a single file. For instance, (as illustrated in Fig. 3) a file containing the meshing instructions for a half model of a shuttle-like orbiter, a half external tank, and a single solid rocket booster can be used with a few cut, paste, and positioning commands to generate a full model of a launch stack, an ascent post SRB separation case, an on-orbit case, or all three as desired.

The user has a wide variety of options for saving the Loft generated mesh. The output module for I-DEAS is the most flexible option. The I-DEAS output module writes an ASCII universal file and supports the changes in the universal file definition for version 7-9 of I-DEAS. I-DEAS version 11 support is in development, although a version 9 file does get read successfully. The NASTRAN bulk data file module is less flexible. Node, element, property, and material cards are written. MSC/PATRAN style comments are used to convey the property names to the later FEA and sizing stages of the process. The VRML 1.0 output module is an excellent tool for quick feedback on the generated mesh. Other modules under consideration for early development include FAST-3D and Lightwave Object formats. The user can use any combination of these output modules at any point in the mesh generation process. Thus, one could save a NASTRAN file containing the fuselage, then an I-DEAS 7 file containing the fuselage and main wing, and finally VRML and I-DEAS 9 files containing the full model.

Loft does have limitations. There is no way to generate meshes for fillets, for strakes, or other preliminary- or later-stage design features. Wing objects are not automatically attached in a finite element sense to the fuselage; rigid elements or boundary conditions need to be added either manually or through algorithm implementation in a controlling program.

Loft is written in standard C. There are executable versions for Windows, UNIX, and Linux.

III HSLOAD: Automated Structural Element Sizing

Structural component sizing, and thus the basis for mass estimation, is performed using a Visual Basic program created called HyperSizerLOAD (HSLOAD). At this point in analysis FEA static solutions have produced elemental loads for a variety of load cases that are thought to be critical to different portions of the vehicle. The sizing stage calculates the beam and stiffened panel thicknesses that are required to carry these loads. Finally the gages and material densities are used to compute a component analysis weight.

Collier Research's HyperSizer is the core of the sizing stage. HyperSizer is an extremely powerful tool for this application, with some usability issues for batch analyses that have been addressed through this use of its user accessible Application Programming Interface (API). This API was developed under previous NASA funding within the NASA High Performance Computing and Communications Program, HPCCP [9].

HyperSizer allows the user to select from a very large range of stiffened panel and open section beam designs. Hundreds of materials are defined in the included database with powerful tools provided for adding more. Dozens of failure criteria are checked for each structural concept. A simple fully-stressed design approach would check material strength; HyperSizer also checks for several types of buckling, crushing of sandwich cores, strain limits if present (in a composite tank for instance), etc.

The HyperSizer user can select a variety of structural concepts, a list of materials, and a range of panel and stiffener sizes. The mathematical combination of these discrete options can produce tens of thousands of candidate designs that need to be tested for failure. HyperSizer sorts these designs by weight and tests each candidate sequentially to find the lowest weight panel that will support all of the defined load sets. It is easy on a large vehicle to create a very large design space that can take unacceptable amounts of CPU time to process. Engineering discretion in design variable selection applied through the HSLoad program improves this performance.

HyperSizer can load a variety of finite element data formats, including the I-DEAS universal file. Program logic automatically collects all elements with the same physical and material properties and combines them into a "component". All elements in a component will share the exact same sizing results. Recall that one of the main reasons for using Loft to generate the finite element mesh was to control the assignment of these properties. The user then collects components into "groups". Group members share the same design space: material options, panel concepts, and design variable limits. But each component in the group may be sized to a different result within that design space. Typical group design space settings can be saved as templates and quickly loaded to set up a group's settings as needed. Another HyperSizer collection is called an "assembly". An assembly is an arbitrary, and not necessarily unique, collection of groups and/or components. Assemblies are primarily used for post-processing purposes such as reporting the weight of major subsystems made of many panels and beams.

An example will help clarify the HyperSizer terminology. Consider a wing consisting of ribs, spars, and skin panels. A reasonably dense mesh will easily consist of a thousand elements or more. Each skin panel (spanning the space between adjacent ribs and spars) is set to be a "component" consisting of a number of finite elements. For an initial rough sizing run all of these skin components may be collected into a single "group". Thus, the entire wing skin would share the same design space settings, but each skin panel would have its own particular sizing result. Finally, the groups containing the skin, ribs, and spars may be collected into an "assembly" to generate a wing subsystem weight.

HyperSizer is typically run interactively and controlled using a graphical user interface. This interface is not well suited for the batch operation required for the current automation process, but, there is an alternative execution approach. HyperSizer has a program accessible API that can be used to control it. This API uses the Microsoft COM standard and requires a compiler that supports that standard. At the recommendation of the HyperSizer creators, Visual Basic 6 was used to create a batch control program for HyperSizer. HSLOAD is that in-house created HyperSizer batch control program. Using instructions it reads from a text input file, it instructs HyperSizer to load the desired universal file, assign components into groups, load pre-generated group templates for the sizing variables, perform a sizing solution, and controls the post-processing analysis of the HyperSizer results.

HSLOAD represents a substantial improvement in efficiency. A very large and detailed launch vehicle model that took three man years to generate had to be loaded into HyperSizer several times due to small changes. The HyperSizer setup time was eventually reduced to one full day. Using HSLOAD, it took a few hours to initially set up the loading instructions in the text file, then an hour of CPU time to set up the HyperSizer analysis of the model. Small changes like considering the trade of metal versus composite fuel tanks took minutes to set up.

HyperSizer has some very powerful graphical post processing tools. It can generate full vehicle plots colored by a very wide variety of solution results. It can also generate extensive documentation of the results in HTML. But, it has no way of just producing a list of components that meet some criteria that the engineer may desire.

HSLOAD is again used in the post-processing stage of the process. It uses the COM API to communicate with HyperSizer and extract a variety of useful information. Some basic items extracted include total vehicle weight as well as the weight of each assembly. HSLOAD also generates text lists of every under- and over-designed component in the analysis. Table 2 shows a partial HSLOAD results file.

A component is considered to have failed or to be under-designed if it has a negative factor of safety. This may mean that the designer needs to increase the allowable thickness of that panel, or change its design in another way such as adding more ribs and spars to a wing to reduce the skin panel size. Since the component names are based on the user's naming convention specified at the meshing stage, it is easy to detect a trend that requires a redesign (e.g. all the root panels on the upper skin of the wing have buckled) or an isolated failure that may indicate a local spike in the loading.

A panel or beam is considered to be over-designed if the lowest weight design has a positive factor of safety. This does not necessarily require engineering intervention. But each of these components represents a possible opportunity to lower the total vehicle weight if their design limits can be lowered. An isolated over-designed component can probably be ignored, but a region of such elements may indicate that it would be profitable to redesign that section of the vehicle (reducing the number of ring frames in a tank for instance).

In the case shown in Table 2, the file gives unit and total weights for the model then lists the under- and over-designed components discovered in the analysis. In this case, there were no failed components and three overdesigned components. By examining the component names, it can be seen that all three are on the root rib of the wing. This can be determined from the description "RIB" and the panel coordinates indicating that they are in span-bay (SB) zero. Here, the user had chosen root boundary conditions that clamped all root rib nodes. This resulted in zero internal loading of the root rib elements and the 100 percent margin of safety shown in the results file.

It should be noted that each of the vehicle modifications suggested to deal with over- and under-designed components can be accomplished with single line edits of the input files for either Loft or HSLOAD.

```
Hypersizer Run Summary
UnitWeightShell: 8.315496
UnitWeightBeam: 0
Total Weight: 7105.52

Total failed/underdesigned
components:0

Overdesigned Components:
  BGWING RIB | SB 0 CB 0  MOS = 100
  BGWING RIB | SB 0 CB 1  MOS = 100
  BGWING RIB | SB 0 CB 2  MOS = 100

Total overdesigned components: 3

Total Components in Analysis:  175
```

Table 2: Partial *HSLOAD* results file

IV JAVA Procedures

LOFT and HSLOAD control major portions of the automated structural design process. They are also however under the control of an executive JAVA program which prepares data, sets up the working directory structure, invokes the commercial FEA application software and generally controls all of the analysis stages chosen to be implemented for a particular structural component, element, or vehicle. This executive program also invokes several JAVA utility functions as required for the problem being defined.

Appendix I shows the listing for a typical one of these control programs called "elv_plf_3dfem.java", a contraction which stands for an "expendable launch vehicle payload fairing modeled with 3d FEA techniques." The steps of this program will be reviewed later in this paper, but the general purpose is to execute LOFT for model creation, execute functions which define loads, boundary conditions and static solution design conditions, execute HSLOAD for structural sizing, and perform a load-path stiffness to element sizing convergence iteration between finite element solutions and HyperSizer element resizing. The fairing component is analyzed subject to 2 pressure loading conditions and is shown in Figure 4.

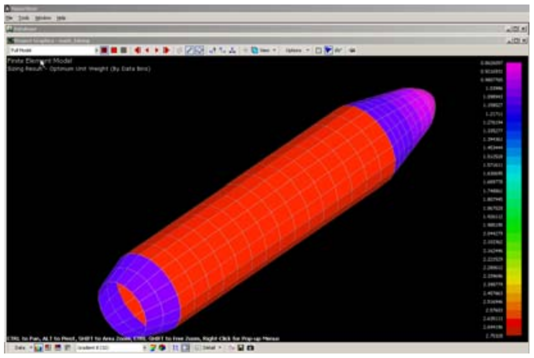


Figure 4. Expendable Launch Vehicle Payload Fairing

Within the control program "elv_plf_3dfem" the JAVA Classes UNV and ModelFile are the means employed to create and modify FEA data. UNV provides access to an ASCII file representation of the finite element data, and ModelFile provides FEA vendor based script file manipulation capabilities to the actual IDEAS FEA binary data. In summary ModelFile allows the user to perform the following generic processes from within the controlling JAVA software:

- read modify and write FEA data
- manipulate finite element groupings of nodes and elements
- create force, pressure and acceleration boundary conditions, also included is the capability to create acceleration vector and fluid density defined head pressures on a partially filled tank.
- create boundary condition restraint data
- combine boundary condition load and restraint data
- Define multiple static solution sets which are the basis for definition of element internal loads used by the HyperSizer analysis step.
- Perform defined static solutions and export model definition and associated results for structural sizing

The terms UNV for universal file, and ModelFile are not generic and come from terminology associated with the IDEAS simulation package. A large part of the data and procedure definitions within these class structures however are defined in more generic terms and it is a further goal to define JAVA procedures whose interface definitions are in terms of non code dependent finite element data. Within the procedure's implementation reside specific code dependent lines of software which perform the desired data manipulations. For example the Class UNV basically provides back to the JAVA environment data structures which characterize the following generic finite element entities and procedures:

nodes	- id, coordinate data, global search routines
elements	- connectivity, material and property callout
props	- property type, defining variables

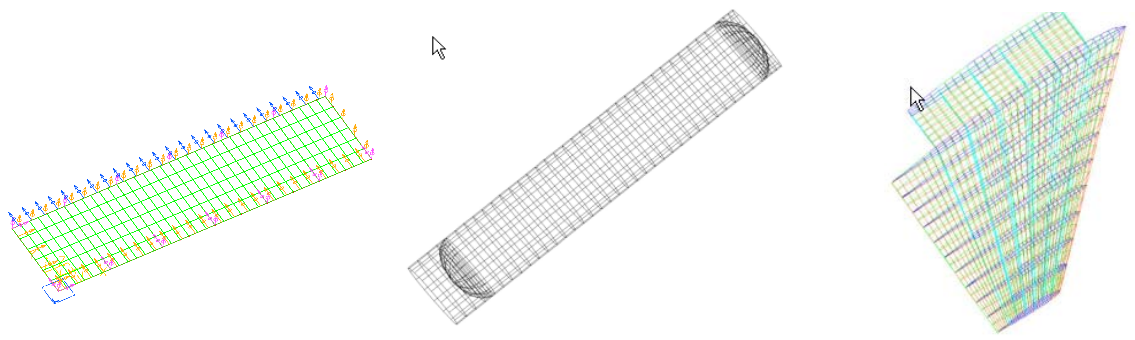
groups	- number of entities in group, associated entity type, and access procedures to element and node arrays
loadsets	- load type, number of loads in set, load values, acceleration data
results	- holds element internal loads and associated element ids

Stepping through Appendix I, program elv_plf_3dfem.java, shows how the use of utility functions within the UNV and ModelFile classes create and analyze the payload fairing. For explanation of the process to people unfamiliar with the JAVA language syntax, the stages of analysis are commented in bold and contained within **/*** … ***/** delimiters. This implementation was done on a personal computer and the first step in the process following the comment **/* Geometry Creation */** executes Loft using a PC batch file process. An existing Loft input is all that is required at this point in time. After Loft execution, program control transfers to the procedure "run_all", where remaining analysis steps are carried out. Within "run_all" we first create a finite element model inside the IDEAS environment and import the universal file FEA data created by Loft. The program developer is now free to combine various functions for his desired loadcase and solution set development. In this instance we first create some groups which can be used for applying pressure boundary conditions. Along with creation of the group "pressure_elements" two pressure loadsets are created, one called "maxq" and one called "meco". After applied loading a restraint group set of nodes is created, in this instance for all nodes between 1325 and 1400 inches in the models X direction. Acceleration vector loads folr the maxq and meco conditions are created and the model is then ready for definition of static solution boundary condition set creation and solution. Solution results in the creation of element level internal loads which are used for structural component sizing. After export of model and results data to an ASCII universal file format, the program is ready to perform structural design utilizing the HSLoad program. An input file for HSLoad is created and the program is called using a batch script in the same manner in which Loft was executed. The remaining calls in elv_plf_3dfem.java perform an iterative analysis between the structural sizing of HyperSizer and the subsequent updated element property (stiffness) model generated by HyperSizer and used for re-execution in IDEAS. This stiffness to loadpath convergence is necessary for statically indeterminate models and is in general a good practice for models of the level of complexity created by LOFT.

Procedures used in Appendix I which create element groups, apply boundary conditions and the like ideally could be defined at an implementation independent generic level such as via a JAVA Class Interface or even more generically via a code interoperability technique such as the IEEE 1516 High Level Architecture (HLA) standard ref [10]. With corporate or even industry standards defined at a high level, a totally interoperable and collaborative design process would be enabled.

V Reference Models

The analysis steps shown in Appendix I for sizing a payload fairing represent a generic process which can be repeated for fairings of different size, aspect ratio, physical construction technique, and loading definition. The modularity of the procedures permit any or all of the above modifications to be made rather easily once the original template program is defined. Also note that new template programs can be made up from the generic modeling and analysis steps provided by the JAVA ModelFile and UNV classes, the program HSLOAD, and the LOFT program. Figure 5 shows some other elements which exist in template format and are readily useable for the study of structural technology, load intensity and component size effects upon structural unit weight and structural component behavior. Figure 6 shows additional reference vehicles under development. These models are termed Reference Models because they are meant to be baseline models which can be studied as is or modified for any of the above reasons. They are also intended to be analyzable by generic finite element and component sizing techniques such that if a common set of data and procedural requirements can be defined the analysis steps are implementable by those who wish to perform the same analyses presented herein, but with alternative software.



Flat Panel Fairing Cryogenic Tank Wing/Empennage

Figure 5 - Typical Reference parts and sub-assemblies

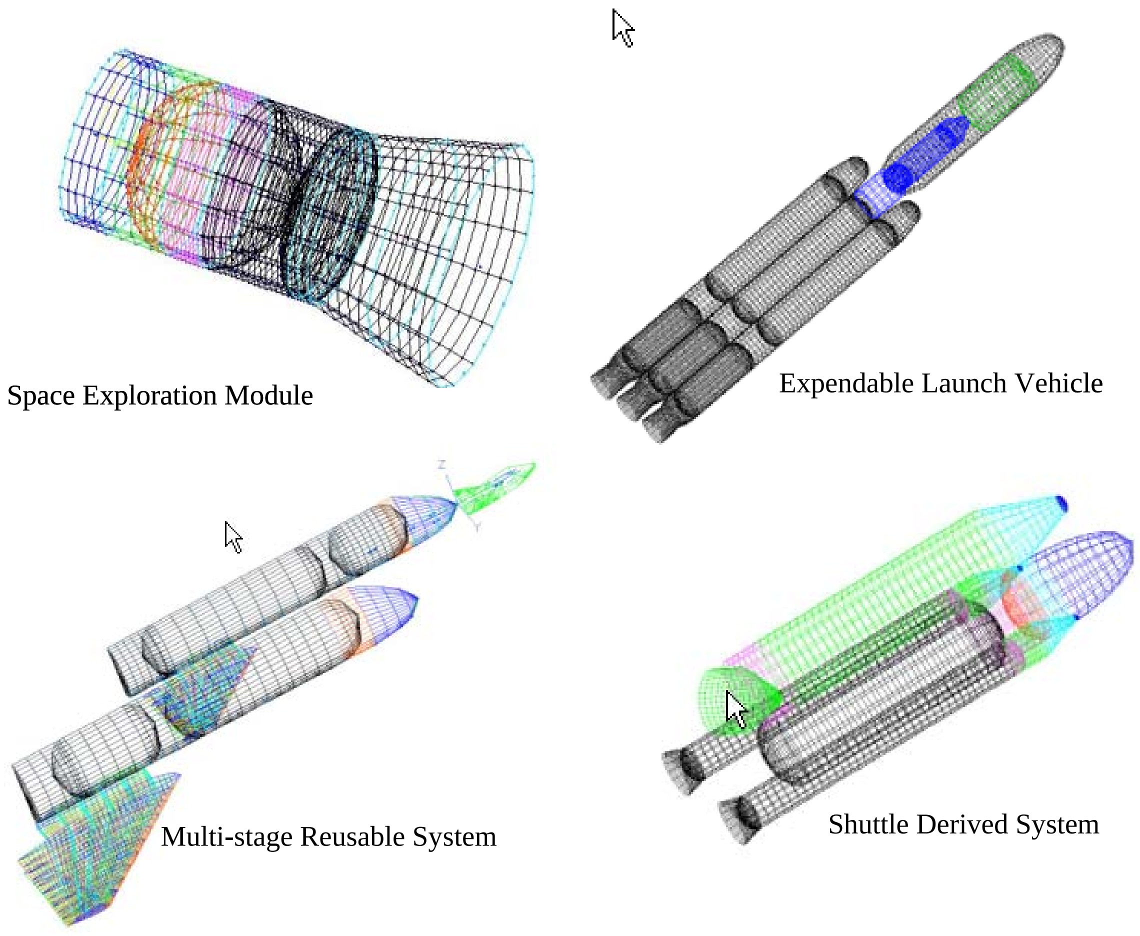


Figure 6 – Reference Vehicles

VI Conclusions

Previous automation techniques for structural analysis and structural mass estimation are improved upon within this paper. The procedures presented can be utilized over a great variety of structural component type and vehicle arrangement. Finite element analysis techniques and commercial structural design software are executed under the control of a central JAVA procedure which is easily modified when parametric and trade study type work is required. Two additional computer programs, Loft for conceptual design mesh generation, and HSLoad for control of the structural sizing process, were created to make the automation process feasible. The analysis data structure, general finite element data, is of complex enough nature such that models and data created at this level of analysis can easily become boundary condition data for more detailed levels of analysis. Simultaneously the Reference Model concept makes available a model library suitable for rapid analysis of components in the more preliminary stages of design. It is this moving of model fidelity from later to earlier phases of design which hopes to increase the sensitivity of system studies to discipline details and thus helps define system architecture and technology requirements which pay off through the deployment and operational phases of a systems life cycle.

A key to creating a successful Simulation Based Acquisition, Simulation Based Design environment for the Government is to have standards defined for the computational work performed in vehicle design and operational assessment. This paper has shown how the practice of performing analysis based structural sizing for concept definition studies can be broken into typical FEA and design steps, and computationally automated. With organizational concurrence the data and data procedures could be standardized in a higher-level defining standard such as HLA. With such standards in place multiple organizations could exchange model data and perform comparative design procedures. Each could be using its own preferred and possibly proprietary computational procedures. Such "plug-and-play" model exchange capability encourages broad industry collaboration. Greater knowledge transfer between system producer and system acquirer permits the acquirer to more quickly define product requirements, and more quickly enables the producer to create the best system level product designs.

References

[1] Cerro, J. A., Martinovic, Z. N., Su, P., and Eldred L.B.: "Structural Weight Estimation for Launch Vehicles", Paper No. 3201, 61st Annual International Conference on Mass Properties Engineering, Virginia Beach, Virginia, May 18-22, 2002.

[2] Taylor, S.J.E., Gan, B. P., Straßburger S., Verbraeck, A.: "HLA-CSPIF Panel on Commercial Off-The-Shelf Distributed Simulation", Proceedings of the 2003 Winter Simulation Conference, pp 881-887.

[3] Kusmik, W.A.: "Optimization in the Simulations Based Design Environment", NAVSEA Division Newport, 1176 Howell Street, Newport, RI 02841.

[4] National Aeronautics and Space Administration: "The Vision for Space Exploration", February 2004, http://www.nasa.gov/pdf/55584main_vision_space_exploration-hi-res.pdf

[5] Eldred L.B.: "Loft, An Automated Mesh Generator For Stiffened Shell Aerospace Vehicles", Swales Aerospace Corporation, NASA LaRC SAMS Contract No. NAS1-00135 03RAA, 5/28/2003.

[6] Eldred L.B.: "HSLoad - Input Deck Documentation", Swales Aerospace Corporation, NASA LaRC SAMS Contract No. NAS1-00135 03RAA, 5/28/2003.

[7] HyperSizer, commercial CAE software, available from, Collier Research Co. 45 Diamond Hill Rd. Hampton, VA 23666

[8] IDEAS, commercial CAD/CAE software, available from, EDS Co. 5400 Legacy Drive, Plano, Texas 75024-3199.

[9] Holcomb L.; Smith, P. and Hunter, P.: "NASA High Performance Computing and Communications Program", NASA-TM-4653, 1994.

[10] Institute of Electrical and Electronic Engineers: "IEEE Standard for Modeling and Simulation (M&S) High Level Architecture HLA - Framework and Rules"

Biographies

Jeffrey A. (Jeff) Cerro:
Education: BS Mechanical Engineering, Rochester Institute of Technology – 1978
MS Mechanical Engineering, Rensselaer Polytechnic Institute – 1981

Mr. Cerro is an Aerospace Technologist in the Exploration Concepts Branch at the NASA Langley Research Center in Hampton Virginia. He is responsible for space access system studies and particularly the influence of structural performance in such studies. His current work centers around analysis of needs for vehicle elements which support NASA's vision for Space Exploration.

Prior to his work in the Vehicle Analysis Branch Mr. Cerro was a structural analyst for the Lockheed-Martin Corporation, where he performed analysis of aircraft and spacecraft structural components.

Zoran Martinovic:
Education: BS. Mechanical & Aeronautical Engineering,
University of Belgrade, Yugoslavia - 1972
MS. Aerospace Engineering, Pennsylvania State University – 1979
Ph.D. Aerospace Engineering,
Virginia Polytechnic Institute and State University - 1987

Dr. Martinovic works with the Exploration Concepts Branch at NASA Langley Research Center and is responsible for structural analyses and trade studies for new concept aerospace vehicles with emphasis on structural weight estimates and technology assessment. His prior experience covers a wide range of subjects related to aerospace vehicles structural engineering in industry, academia and research.

Lloyd B. Eldred:
Education: B.S. Aerospace and Ocean Engineering,
Virginia Polytechnic Institute and State University - 1986
M.S. Aerospace Engineering,
Virginia Polytechnic Institute and State University - 1989
Ph.D. Aerospace Engineering,
Virginia Polytechnic Institute and State University - 1993

Dr. Eldred is a Senior Engineer for Swales Aerospace Corporation in Hampton Virginia. His work at NASA, LaRC has focused on tool development for improving structural weight estimation of launch vehicles. He has previously worked as a contractor at NASA, Ames Research Center and at Wright-Patterson AFB. He has just accepted a new position with Northrup Grumman, Newport News where he will be performing structural analyses on the next generation aircraft carrier.

Appendix I

JAVA Control Program for Sizing of an ELV Payload Fairing

```
public class elv_plf_3dfem {
  public Model_File mf;
  public consiz2unv_whole_ms9 cs2unv;
  public String ideas_project;
  public String ideas_mf;
  public String work_dir;

/* constructor */
  public elv_plf_3dfem() {
    try {
      mf = new Model_File();
      cs2unv = new consiz2unv_whole_ms9();
    } //catch (IOException e) {
    //System.out.println("IOexcepted" + e);
    //}
    finally {
    }
  } // end constructor

    public static void main (String args[]) {
    elv_plf_3dfem a = new elv_plf_3dfem();
      a.num_des_solsets=2;

      /* Geometry Creation */
      try {
       ExecDemo1 bat_file = new ExecDemo1();
       bat_file.run_bat_file("F:\\ref_part_elv_fairing\\run_loft.bat","F:\\ref_part_elv_fairing");
       }
       catch (Exception exc1) {
         String err = exc1.toString();
         System.out.println(err);
       }
        a.run_all("cerro", // project, existing
              "F:\\ref_part_elv_fairing\\elv_fairing.mf1", // model file, existing
              "F:\\ref_part_elv_fairing\\", // workdir, existing
              "MarkFairing.unv", // unv (may be existing, will get overwritten)
              "C:\\Program Files\\Microsoft Office\\Office\\EXCEL.EXE"); //excel command
    } // main
```

```
/* most execution stages are within the run_all procedure */
public void run_all(String ideas_project, String ideas_mf,
                    String work_dir, String unv_filename, String xl_cmd) {
  this.ideas_project = ideas_project;
  this.ideas_mf = ideas_mf;
  this.work_dir = work_dir;
  elv_plf_3dfem a = new elv_plf_3dfem();
  String unvfn = work_dir + unv_filename;

    /* create an IDEAS modelfile from the LOFT created universal file */
     a.mf.set_units("inch", work_dir + "setunits.prg");
     a.mf.run_prg(ideas_project, this.ideas_mf, "simulation", "meshing",
                  work_dir + "setunits.prg");

     a.mf.import_unv(unvfn, work_dir + "imp_unv.prg");
     a.mf.run_prg(ideas_project, this.ideas_mf, "simulation", "meshing",
                  work_dir + "imp_unv.prg");

      /* create group  for and apply pressure loading */
      a.mf.create_group("pressure_elements", work_dir + "group_elems.prg");
      a.mf.run_prg(ideas_project, ideas_mf, "simulation", "bo",
                   work_dir + "group_elems.prg");
      a.mf.add_all_elems_to_group(17, work_dir + "els_to_group.prg");
      a.mf.run_prg(ideas_project, ideas_mf, "simulation", "bo",
                   work_dir + "els_to_group.prg");

      a.mf.create_appl_press_loadset_v9("maxq", 17,
                                  -6.18, work_dir + "maxq_pres.prg");
      a.mf.run_prg(ideas_project, ideas_mf, "simulation", "bo",
                   work_dir + "maxq_pres.prg");

      a.mf.create_appl_press_loadset_v9("meco", 17,
                                  -.618, work_dir + "meco_pres.prg");
      a.mf.run_prg(ideas_project, ideas_mf, "simulation", "bo",
                   work_dir + "meco_pres.prg");

      /* create restraint group and add clamped constraint at one end of model */
      a.mf.create_group("clamped_nodes", work_dir + "group_nodes.prg");
      a.mf.run_prg(ideas_project, ideas_mf, "simulation", "bo",
                   work_dir + "group_nodes.prg");

      try {
        BufferedReader infile = new BufferedReader(new FileReader(unvfn));
        BufferedReader in = new BufferedReader(new InputStreamReader(System.
            in));
    File outputFile = new File(work_dir + "debug_unv.txt");
    FileWriter out = new FileWriter(outputFile);
    unv_ms9 unv_inst = new unv_ms9(infile, out);
    infile.close();
    // add nodes to restraint group
    int group_id = 18;
    int[] res_nodes = unv_inst.nodes.nodes_in_xrange(1325., 1400.);
    a.mf.add_to_group("nodes", res_nodes, group_id, work_dir + "nodes_to_group.prg");
    a.mf.run_prg(ideas_project, ideas_mf, "simulation", "bo",
                 work_dir + "nodes_to_group.prg");
    // create restraint case for clamp condition
    String[] str = {
        "AF", "YT AT FI"};
    String[] str2 = {
        "CLAMP"};
    a.mf.create_restraint_set("clamp", res_nodes, str2,
                              work_dir + "res_set.prg");
    a.mf.run_prg(ideas_project, ideas_mf, "simulation", "bo",
                 work_dir + "res_set.prg");
  }
  catch (IOException e) {
    System.out.println("IOexcepted" + e);
  }
/* add acceleration loads */
 //  method to apply acceleration to the design condition loadsets
 if (true) { // add accel vectors to loadsets
   // add the appropriate acceleration
   double[][] accel = {
       {-386.4 * 2.3, .0, 386.4 * 2.0}, // maxQ
       {-386.4*6.0, .0, 386.4*0.5} // meco
   };

   double ax, ay, az;
   int no_sets = 2;
   for (int i = 0; i < no_sets; i++) {
     ax = accel[i][0];
     ay = accel[i][1];
     az = accel[i][2];
     int[] ldsetno = {1,2};
     a.mf.add_accel_to_loadset(ldsetno[i], ax, ay, az,
                               work_dir + "add_accel.prg");
     a.mf.run_prg(ideas_project, ideas_mf, "simulation", "bo",
                  work_dir + "add_accel.prg");
   }
 }
```

```
/* create the boundary condition sets for use in HyperSizer */
  int[] ldsetids = {1};
  a.mf.create_bc_set("maxq", 1, ldsetids,
                  "clamp", work_dir + "create_bcset.prg");
  a.mf.run_prg(ideas_project, ideas_mf, "simulation", "bo",
             work_dir + "create_bcset.prg");
  ldsetids[0]=2;
  a.mf.create_bc_set("meco", 1, ldsetids,
                  "clamp", work_dir + "create_bcset2.prg");
  a.mf.run_prg(ideas_project, ideas_mf, "simulation", "bo",
             work_dir + "create_bcset2.prg");
/* and solve the above bc sets */
  // solve for each design loadset
  // sol set name, bcset no, runtitle, list fn, prg fn

  a.mf.create_sol_set_solve("maxq", 2, "maxq_run",
                        work_dir + "maxq.lis",
                        work_dir + "solve.prg");
  a.mf.run_prg(ideas_project, ideas_mf, "simulation", "MO",
             work_dir + "solve.prg");

  a.mf.create_sol_set_solve("meco", 3, "meco",
                        work_dir + "meco.lis",
                        work_dir + "solve2.prg");
  a.mf.run_prg(ideas_project, ideas_mf, "simulation", "MO",
             work_dir + "solve2.prg");

  /* export universal file of model and results for use in HyperSizer, HSLoad */
  a.mf.create_unv_all(unvfn, work_dir + "createunv.prg");
  a.mf.run_prg(ideas_project, ideas_mf, "simulation", "meshing",
             work_dir + "createunv.prg");

/* setup and run HSLoad */
  StringTokenizer s = new StringTokenizer(unv_filename, ".");
  String fn_prefix = s.nextToken();
  hs_load b = new hs_load();
  int[] mech_ls_number = {
     1};
  int[] thermal_ls_number = {
     0};
  String[] ls_name = {
     "maxQ"};
  String[] flag = {
     "true"};
  String[] group_name = {
    "mygroup1"};
  String[] family_name = {
    "Sandwich"};
  String[] template_name = {
    "zoran_im7_foam_sandwich"};
  String[] component_match = {
    "nosetip", "nosecone", "barrel", "boattail"};
  b.write_input_file(work_dir + "HSLoad_solve.in",
                  "mark_fairing",
                  "c:\\program files\\hypersizer",
                  "c:\\hypersizerfea",
                  "working1_4.0.hdb",
                  "solve",
                  unvfn,
                  unvfn,
                  work_dir + fn_prefix + ".pm1",
                  1,
                  mech_ls_number,
                  thermal_ls_number,
                  ls_name,
                  flag,
                  1,
                  group_name,
                  family_name,
                  template_name,
                  component_match);
}
  try {
    ExecDemo1 bat_file = new ExecDemo1();
    bat_file.run_bat_file(work_dir + "run_HSLoad.bat", work_dir);
  }
  catch (Exception exc1) {
    String err = exc1.toString();
    System.out.println(err);
  }
 /* Iterate structural stiffness-loadpath and resizing between hypersizer and FEA
    solutions    */
  for (int i = 1; i <= 2; i++) {
    System.out.println(" iteration " + i + "\n");
    a.mf.delete_fem("fem_is_item3", work_dir + "del_fem_item3.prg");
    a.mf.run_prg(ideas_project, this.ideas_mf, "simulation", "meshing",
              work_dir + "del_fem_item3.prg");

    a.mf.set_units("inch", work_dir + "a.prg");
    a.mf.run_prg(ideas_project, this.ideas_mf, "simulation", "meshing",
                work_dir + "a.prg");

        a.mf.import_unv(work_dir + "mark_fairing-HS.unv", work_dir + "a.prg");
        a.mf.run_prg(ideas_project, this.ideas_mf, "simulation", "meshing",
                work_dir + "a.prg");

        a.mf.assign_part_material(work_dir + "assign.prg");
        a.mf.run_prg(ideas_project, this.ideas_mf, "simulation", "meshing",
                work_dir + "assign.prg");

        a.mf.run_static_sol(1,"maxq_run",work_dir + "maxq.lis",
                   work_dir + "solve1.prg");
        a.mf.run_prg(ideas_project, this.ideas_mf, "simulation", "meshing",
                work_dir + "solve1.prg");

        a.mf.run_static_sol(2,"meco_run",work_dir + "meco.lis",
               work_dir + "solve2.prg");
        a.mf.run_prg(ideas_project, this.ideas_mf, "simulation", "meshing",
          work_dir + "solve2.prg");

        a.mf.create_unv_all(unvfn, work_dir + "createunv.prg");
        a.mf.run_prg(ideas_project, ideas_mf, "simulation", "meshing",
                work_dir + "createunv.prg");
        try {
          ExecDemo1 bat_file = new ExecDemo1();
          bat_file.run_bat_file(work_dir + "run_HSLoad.bat", work_dir);
          java.lang.Runtime rto = java.lang.Runtime.getRuntime();
          String cmd = "copy mark_fairing.hsresult mark_fairing.hsresult" + i;
          String[] cmds = {
              cmd, work_dir};
          Process p1 = rto.exec(cmds);
          p1.waitFor();
        }
        catch (Exception exc1) {
          String err = exc1.toString();
          System.out.println(err);
        }
    } // run_all

} // class
```